# LEGIBILITY NOTICE

A major purpose of the Technical Information Center is to provide the broadest dissemination possible of information contained in DOE's Research and Development Reports to business, industry, the academic community, and federal, state and local governments.

Although a small portion of this report is not reproducible, it is being made available to expedite the availability of information on the research discussed herein.

**LA-UR** -89-1693

Los Alamos National Laboratory is operated by the University of California for the United States Department of Energy under contract W-7405-ENG-36

LA-UR--89-1693

DE89 013442

TITLE XCL - A FAMILY OF PROGRAMMING LANGUAGE-BASED SHELLS

AUTHOR(S) Mark A. Roschke, C-10

SUBMITTED TO Cray Users Group Meeting, Universal City, California, April 24-28, 1989

## DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

Los Alamos

Los Alamos National Laboratory
Los Alamos, New Mexico 87545

# XCL - A Family of Programming Language-Based Shells

Mark A Roschke*
Computing and Communications Division
Los Alamos National Laboratory
Los Alamos, NM 87545

## Introduction

As the three major UNIX shells have emerged, they have shown little inclination to include syntax and semantics from existing programming languages. The first of these shells, *sh* [1], contains only a small amount of C-like syntax. *Csh* [2] provides some C language expression syntax, but includes very little other C syntax. The newest of these shells, *ksh* [3], also includes some C-like expression syntax, although they contain some significantly un-C-like syntax in such areas as relational and logical operators.

Several much less widely used shells [4-6] have been written that much more closely resemble particular programming languages. However, each of these shells have the disadvantage of not being based upon a widely used programming language. In addition, interactive commands tend to be difficult to enter because they must frequently be entered using normal programming language constructs such as function calls.

Thus, the vast majority of programmers of today's UNIX shells must deal with a shell interface that is not based upon any familiar programming language. This paper describes some of the features of the *xcl* family of shells. Each of these shells is based closely upon an existing programming language and provides the user with a familiar and highly programmable shell interface

## The Familiar Programming Language

Only a very few users of today's UNIX shells are provided with a shell interface based upon a familiar programming language. In contrast with current UNIX shells, the *xcl* family of shells provides the user with a series of shell interfaces, each of which is based heavily upon a widely used programming language.

*Xcl* is based upon two fundamental premises that a programming language can serve as a good workable basis for a shell, and that it is highly desirable for that programming language to be one with which the user is familiar

Even though the primary object of a shell is to execute processes, much of the code written in a shell involves concepts found in all of today's programming languages These concepts include such areas as looping, decision making, string manipulation function execution and arithmetic calculations

Basing a shell closely upon a familiar programming language provides several major advantages to the user. First, much less time and effort is required on the user's part to learn a sufficient portion of the shell in order to be productive. Second, a user can make use of current programming habits and problem solving techniques by being able to use familiar everyday constructs as block structures, expressions, and library functions.

The more comfortable the user is with the capabilities of the shell, the less inclination there is for the user to resort to writing code in a programming language to accomplish a given task that may be well suited to a shell. This is especially important when efficiency is not of primary importance, since shell algorithms often consist of 10 to 100 times less code than the same algorithm coded in a programming language.

The requirement that the programming language must be familiar generates its own implementation challenges. If a shell is based only upon a single programming language, those users programming in any another language are essentially ignored. The only way to provide a shell based upon a familiar programming language (short of requiring all programmers to program in the same language) is to provide a series of shells based upon different programming languages. *Xcl* provides shell interfaces based upon the C and FORTRAN programming languages, with the basic design of the shells allowing for relatively easy implementation of other language interfaces

## Language Features

The features directly related to the respective programming language cover a wide range of syntax and semantics, thereby providing the user with a much larger familiar language environment than can be obtained by providing only a limited set of familiar language constructs, such as only expressions

Since any deviation from the syntax and semantics of the respective programming language increases the potential for confusion mistakes, and frustration, such deviations have been kept to a minimum and are largely confined to the area of declarations (or lack thereof)

## Programs and Functions

Current shells have essentially no equivalent to functions in the programming language sense of the word and instead provide a

*This work was performed under the auspices of the U. S. Department of Energy facility that amounts to a single-function program. Although the term "function" may even be used to refer to such a facility, the way in which these facilities interact with each other has most, if not all, of the following important attributes of a program facility: command line syntax, completion status, and the propagation of environment and signal values. As a result, the user is prevented from utilizing a true function facility (in the programming language sense of the word) when approaching any given problem, and the resulting code is often far from optimal because of such a limitation. In addition, the look and feel of these single-function programs is further complicated by the fact that most shells globally scope (at least by default) user variables across these single-function programs.

The inclusion of both programs and functions in *xcl* has several advantages. First, the user may take advantage of such familiar function features as formal arguments, explicitly set and referenced return values, and familiar inter-function scoping rules. Second, standard library functions such as I/O, string manipulation, and informational functions can be provided, and other library functions can be provided in a way that naturally extends the capabilities of the language. And third, the inclusion of a function facility then allows for a separate program facility that has the normal features of inter-program communication, including not scoping user variables across program boundaries.

### Block Structures and Expressions

Much of the block structure and expression syntax of *xcl* is taken directly from the respective programming language. This provides the user with a powerful set of familiar capabilities with which to approach programming problems. In those few instances in which these capabilities are insufficient for a shell environment, such as in loops and string manipulation, several loop structures and string operators were added.

In addition to the normal use of expressions in programming language constructs, *xcl* also allows language expressions to be placed in non-language commands by simply enclosing the expressions within braces. This feature is referred to as expression substitution and is a much more powerful feature than the forms of parameter substitution employed by today's major UNIX shells. (Non-language commands are those commands that have a command-line style syntax and are used to execute processes or built-in commands). This allows the programmer to use essentially the same programming language expression syntax in virtually any command without ambiguity.

### Commands

Many *xcl* commands are drawn directly from the respective programming language and very closely follow the syntax of the programming language. In keeping with the concept of consistent syntax, these programming language-based commands are not marked with any special syntax in order to differentiate them from other *xcl* commands. And, in keeping with programming language conventions, programming language-based commands do not have any completion status associated with them.

### Variables

There are three types of variables in *xcl*: user variables, environment variables, and control variables. User variables follow a familiar scoping convention in that they are local unless declared to be global.

Environment variables are directly patterned after UNIX environment variables. They follow the same scoping rules as UNIX; i.e., they are propagated across program boundaries in a downward-only fashion.

Control variables may be thought of as a kind of *xcl* control panel having many switches and dials that control the internal execution of *xcl*. These variables contain such values as prompts, search paths, and various substitution flags. Although they have no direct programming language counterpart, they may still be referenced in the same way as the other two types of variable described above, and they are global in scope across functions.

### Debugging

Even in current shells there is significant room for improvement in the area of debugging. But when the programmer is presented with a shell based closely upon a familiar programming language, significantly more complex shell programs are written, which further increases the need for improved debugging capabilities.

*Xcl* provides a dynamic debugging facility based upon the *dbx* [7] debugger. This facility includes a significant subset of the *dbx* interface, allowing the user to set breakpoints, print the value of variables or expressions, etc. This debugging facility can drastically reduce the amount of time needed to debug a shell program, and using familiar debugging commands drastically reduces the amount of time needed to learn to use such a facility. Each shell version allows expressions to be entered using the syntax of the respective programming language.

### Language Differences

*Xcl* diverges from the respective programming syntax in several areas, the first of which is the type of variables. For all practical purposes, all variables may be thought of as being of type string. This approach is much more conducive to the manipulation of text messages than is a strongly typed implementation. All operands are interpreted in the context of the applied operation. Thus, two strings may be concatenated or added, and only in the latter case are the operands required to be numeric. And the result of an arithmetic operation involving different type operands is always converted to the type of the predominant operand. For example, in the case of an integer added to a floating point number, the result would be a floating point number

The second area involves storage reservation. No declarations defining maximum string size are used. Instead, each value is assigned memory as needed during execution.

The third area of divergence involves subscripts. Arrays of strings are not directly supported. Instead, subscripts are included to provide a convenient method of referencing individual words or characters within strings. (A word is a substring delimited by unquoted white space.)

### Non-Language Commands

Non-language commands are those commands that are not based upon the respective programming language. These commands have a command-line syntax and may be interspersed with the language-based commands. These non-language commands provide the user with easy-to-enter (blank delimited) commands. They also provide the user with a familiar set of powerful operators. The operators deal mostly with various forms of substitution, such as filename, home directory, and command substitution. Since there is inevitable overlap in syntax between language and non-language commands, allowing both of these command types provides the user with both of these indispensable sets of capabilities without being confronted with an unfamiliar syntax.

A disavantage of this approach is that process executions cannot be used directly as logical expressions. In order to achieve such an effect, process executions must be performed as function calls. However, this disadvantage was considered to be a relatively small price to pay for all of the language capabilities that are gained as a result of such an approach to the design of the command language.

### Example 1

The following FORTRAN interface example is an *xcl* routine to set the command prompt. The input argument to this routine is a directory name. The {...} sequences are similar to parameter substitution found in the three major UNIX shells, except that any arbitrary language expression may be placed within the braces. The prompt is set to the name of the host computer, followed by the last thirty characters of the directory name, followed by '% '.

```
subroutine setprompt(dir)
n = len(dir)
if (n .gt. 30) dir = dir(n-29:30)
set prompt "{gethostname()} {dir} % "
return
end
```

### Example 2

The following C interface example is an *xcl* routine to find a file in the standard search path. The cmdline command allows the routine to be passed a command line instead of formal arguments. In this case, only a single file name is allowed. Its value is automatically placed into the local variable named file before the routine begins execution. The function strlenw is used to count the number of words in the path control variable, which is syntactically marked with a leading '$'. And the // operator concatenates two strings.

```
findpath()
{
        cmdline -name file -position 1
        for (i=0; i<strlenw($path); i++)
        {
                if (strcmp(file, $path[i]) == 0)
                        puts($path[i] // "/" // file);
        }
}
```

### Current State

*Xcl* is written in C and runs under BSD 4.3 UNIX. Future plans include a System V version, followed by a UNICOS version. A number of major features have not yet been implemented, including foreground/background process control, and a screen editing style of command re-execution.

### Summary

*Xcl* is a series of shells that can significantly increase user productivity. This increase is accomplished by providing familiar programming language syntax and capabilities while retaining proven traditional interactive features found in the major UNIX shells.

### References

[1] Bourne, S. R., "An introduction to the UNIX shell", *UNIX Programmer's Manual, vol. 2A*, 7th ed., Bell Telephone Laboratories, Murray Hill, N.J., Jan. 1979.

[2] Joy, W., "An introduction to the C shell", *UNIX Programmer's Manual, vol. 2C, part 1, Virtual VAX-11 Version*, Computer Systems Research Group, Dept. of Electrical Engineering and Computer Science, Univ. of California at Berkeley, Aug. 1983.

[3] D. G. Korn, "KSH - a shell programming language", *Proceedings of the 1983 Summer USENIX Conference*, Toronto, pp. 191-202.

[4] M. Matthews and Y. Kamath, "The FP-shell", *Proceedings of the 1984 Summer USENIX Conference*, Salt Lake City, Utah, June 1984, pp. 133-140.

[5] C. S. Macdonald, "fsh - A Functional UNIX Command Interpreter", *Software - Practice and Experience*, 1[illegible] pp. 685-700.

[6] J. R. Ellis, "A Lisp Shell", *SIGPLAN Notices* Vol. [illegible] No. 5 (May 1980), pp. 24-34.

[7] DBX (1), *UNIX User's Manual, Reference Guide*, [illegible] Berkeley Software Distribution, Computer Sc

Division, University of California, Berkeley, Calif, March 1984.